游易-程序主题分享合辑

+ 收藏专题

知识管理部 等 2022.06.07 15:37 6236 321 184个资源

汇总从101至今的游易征稿文章合辑。

专题 游易程序第125期 手游内存优化

目录

浅谈python内存优化手段

贾治中 2019.10.08 13:09 1514 20 0 查看原文

本文仅面向以下用户开放，请注意内容保密范围

查看权限：互娱正式-公开

> 内存优化在公司项目的实际使用中早已屡见不鲜了，本文主要就过去项目中使用的内存优化方法进行一个总结。

内存优化在公司项目的实际使用中早已屡见不鲜了，本文主要就过去项目中使用的内存优化方法进行一个总结。大致内容如图1所示，如

本文python默认版本是2.7。

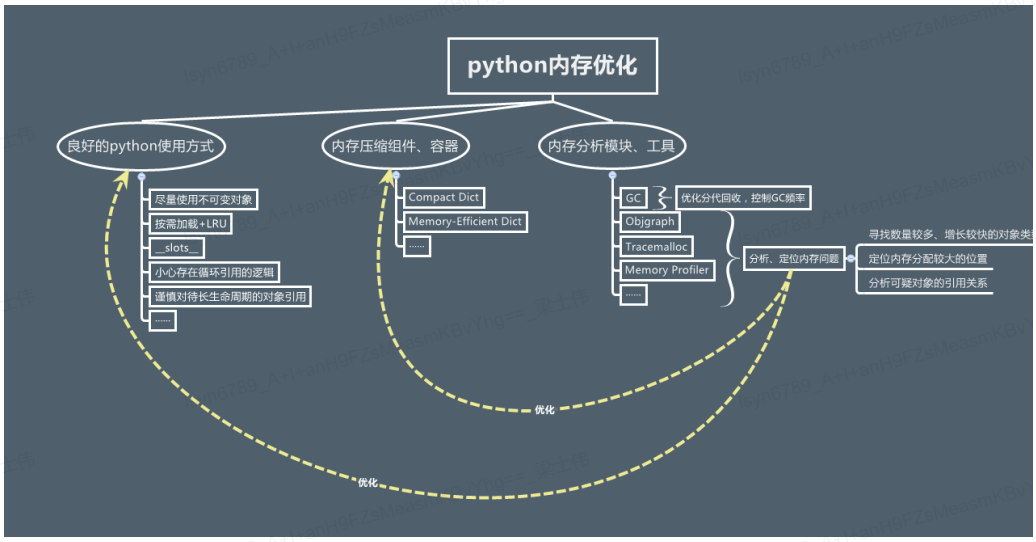

图 1 内存优化思路图

1 Effective Python

1）**尽量使用immutable对象。**尽可能使用迭代器或itertools中相关的ixxxx方法，减少容器对象创建；使用tuple替代list；zens set；使用生成器等。

2）**尽量不要在模块作用域中写逻辑，对全局范围内的临时变量进行优化。**

3）**按需加载+LRU。**

4）**使用__slots__。**python自定义对象一般使用__dict__存储属性，然而dict类型所占空间一般要比实际占用的空间大。而在新式类中声__slots__，就会提前确定好这个类的object会有哪些成员属性，python虚拟机会预留对应的内存。同时，这样的object将不会有__dict_性，也不能动态添加新的属性。KM上也有相应的文章介绍过slots这种特性，有人使用slots进行内存优化节省了9G内存（相关文章）。

公司大多数游戏使用的数据表都是基于python dict的，dict在python虚拟机中用到的地方非常多，其设计决定了随着dict中数据的增多空间也随之增大。各个项目组也用了很多方法去压缩这个数据dict，包括DesignData, TupleDict, SparseDict, TaggedDict，基本思路是**用immutable对象或优化hash算法**，具体可以参考Python导表数据处理和内存优化策略，游戏数据的内存优化尝试。目前项目中用的应该是TaggedDict，其思想主要利用key、value指针尾部0位存储hash冲突时下一个查找位置。蒋宇翔同学提出的基于差异控制的数据使得客户端数据表的访问形成一种database+cache的模式，从内容上着手使数据表内存相对比较紧凑。

3 Memory-Efficient Dict

超哥在A Memory-Efficient Implementation of Python Dict Objects一文中提出了一种更高效的改进CPython dict的实现（python基于这一思路实现了更加高效的dict，这种split dict主要是用于优化实例对象上存储属性的tp_dict，设计思路和实现方式见PEP 412 -- Sharing Dictionary）。

主要思想

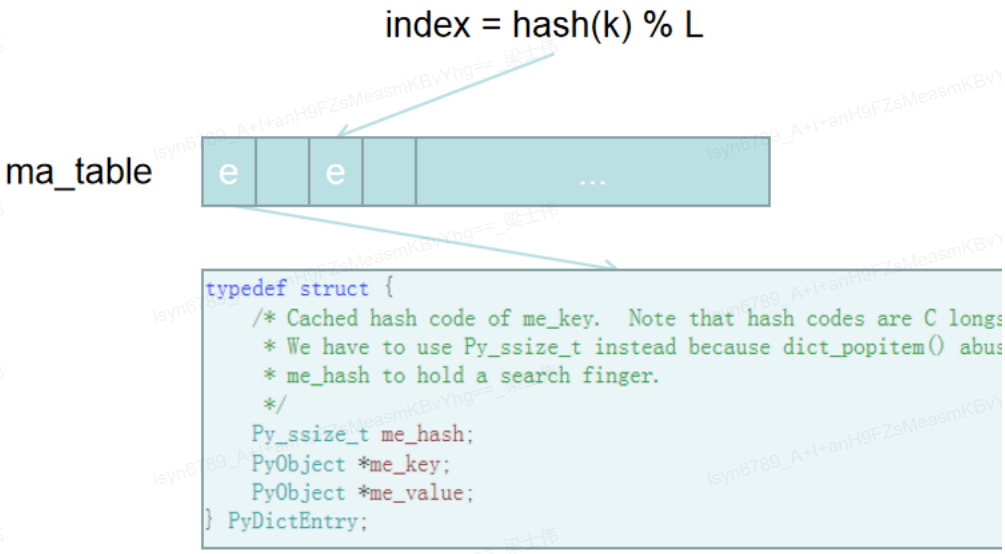

图 2 原生实现

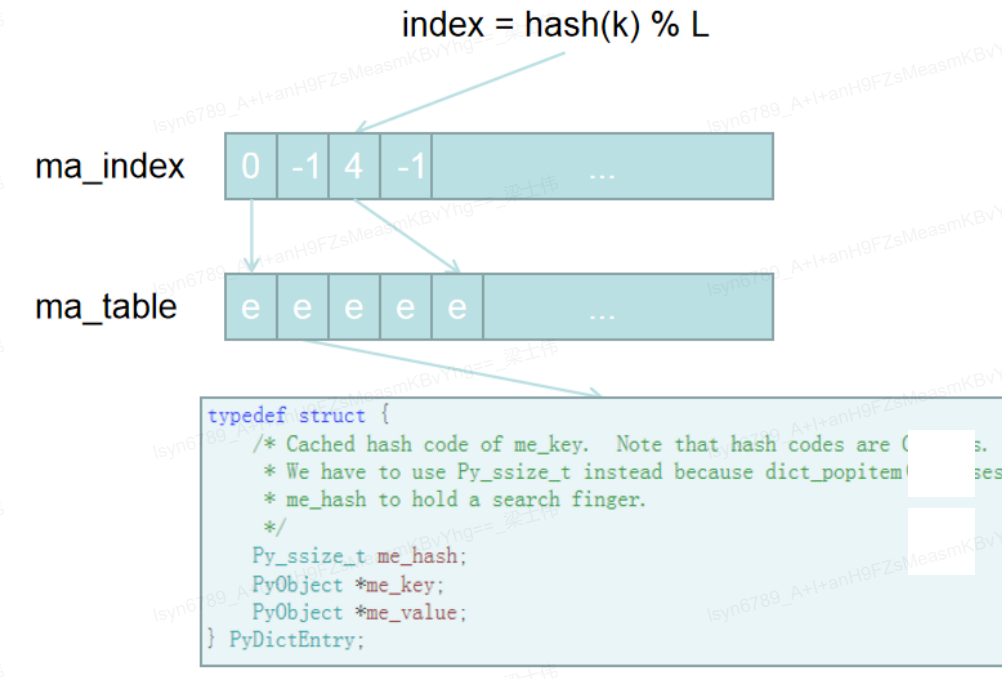

图 3 改进方案

考虑到游戏中用到的dict大小一般都不会太大，可以进一步缩小index_table占用空间。对于元素个数小于128的使用Byte下标，个数小于的使用short integer。因为此处存储的下标是有符号整数，-1表示空元素。（为什么不直接使用无符号整数下标呢？这样可以扩大索引要用0表示空元素，同时ma_index中的下标访问ma_table时下标减一就好了。如此就可以使用Byte下标表示255个元素，short integer65535个元素。但事实上，负数还有其他用处，dict中被删除的元素可以用-2表示，以及未来一些需要扩展的地方）。

好处

1）**更快的迭代访问速度。**老的实现会在ma_table上访问很多空的元素，新的方案可以直接在ma_table上线性访问所有元素，较少碰到（一般是删除了的dummy元素）。

2）**天生支持有序插入。**新方案天生就是OrderedDict，且效率更高，占用空间更小。

4 GC

python的内存管理策略主要是**引用计数、垃圾回收GC（标记清除和分代回收）。**除了在小对象缓冲池中的对象（如部分整数对象（-5~被intern机制处理过的少于20个字符长度的字符串）外，引用计数变成0的对象会直接被删除。对于因循环引用而导致的引用计数不为0（只有容器对象才会出现循环引用）则会使用垃圾回收机制，为了减少或避免游戏进程中GC带来的stop the world，项目中一般会使用略：

提高分代回收阈值

分代回收是一种**以空间换时间**的策略，基本思想是将对象按照其存活时间存放在不同的集合（代），存活时间越长的代，其中对象的垃圾率越低。可以这么理解，**经过越多的垃圾回收次数而存活下来的对象，就应该降低其垃圾回收的频率。**python中对象一般分为三代存放链表中，0、1、2代回收的阈值默认是700,10,10。每次容器对象分配内存的时候，generation[0].count都会加1，一旦generation[0].c700，就会立即出发垃圾回收。

分代回收的过程如下：

1）从年老代开始往前遍历，找到第一个代中count超过当前代阈值的代cur_generation。

2）将比cur_generation更老的一代（如果存在）的count加1，将cur_generation及更年轻的代的count置0。

3）将比cur_generation更年轻的代的对象链表合并到cur_generation中去。

4）对cur_generation中的对象链表进行标记清除算法，找到unreachable objects，此时代中剩下的都是reachable objects。

5）将cur_generation中对象并入更老的一代（如果存在）。

6）对unreachable链表中的对象进行处理，包括弱引用对象（调用callback）和定义了__del__函数的对象（这些对象及其引用的对象都收，加入finalizers链表，gc.garbage可以获取到，最后也会加入到更老的一代中）。

7）对unreachable链表中剩下的对象进行回收。

所以，为了降低垃圾回收的频率，我们可以通过**gc.set_threshold**操作调大每一代的回收阈值，这个阈值需要根据游戏实际情况调配出是通过给第三代GC（一般是存放数据表之类的对象）设置一个很大的值来关闭full GC的，徐星同学也有比较详细的关于改造分代回收的

手动开启垃圾回收

通过**gc.disable()**可以禁用python的自动垃圾回收，我们可以选择在游戏进程运行的合适时机通过**gc.collect(n)**手动进行垃圾回收建大量对象的时候开启GC，n表示回收的代）。比如可以在客户端玩家loading副本或传送的时候开启一次垃圾回收；客户端在战斗前关收，战斗结束后再开启；服务器在线reload的时候关闭GC。

完全关闭垃圾回收

实在不想受GC STW干扰的话，可以完全关闭GC，不过要保证游戏进程中unreachable的对象不存在循环引用。程序中循环引用往往难只要在**适当的时候解开循环或使用weakref**就好了。但大多数时候循环引用是不经意间造成的，这个时候就需要我们去定期监控游戏的（项目中后期使用bot去压测服务器或客户端相关流程，将每个测试用例的内存占用导出到文件中进行后期分析），排查泄露的可能。一存处于非正常增长状态时，就可以使用下文介绍的工具进行分析。

5 Objgraph

出现内存泄露的对象大体分为两种：**存在循环引用的对象**（关闭GC的情况）和**被生命周期较长的对象引用的无效对象**（比如在线玩家A还保留着对下线玩家NetworkConnection组件的引用或来自C++模块的引用等）。objgraph是一款非常好用的**针对python对象类型进内存分析工具，能够生成对象间的引用关系图**（可以用graphviz这个软件绘制出来）。

在实际项目中，有下面几个比较有用的接口：

objgraph.show_most_common_types(n)可以列出数目倒排序处于前n个的对象类型，一般而言都是dict、list、tuple on对象比较多。通过这个接口我们可以找到数目较多的自建类型，如果某种类型对象（比如NetworkConnection）的数目和 线性们就可以通过对比确认其是否泄漏。同时，对于数量较多的自建对象，如果其属性一般固定不变的话，可以使用__slots__进行内存优化。

objgraph.show_growth(...)可以找到自上次调用这个函数以来增长较多的前n个对象类型及对应的增加数，一般可以用来定位异常增类型。

objgraph.by_type(type_name)可以根据对象类型字符串获取该类型的对象列表。

objgraph.show_refs(...), objgraph.show_backrefs(...)都可以把对象的引用关系导成图文件，但是如果对象之间的引用关系比较复杂，读性不好。这时候可以用使用objgraph.find_backref_chain(...), objgraph.show_chain(...)**找到符合条件的节点构成的引用链（最短引

2）在特定的时间点（如进入副本或PVP、离开副本或PVP），使用objgraph.show_growth(...)获取增长较多的对象类型，查看是否有异常对象，若有转到3)。

3）对数量异常或增长异常的对象，使用objgraph.show_refs(...), objgraph.show_backrefs(...)寻找其引用关系，然后去代码中排查是非还是循环引用导致的。

示例代码1：

```
import gc
import objgraph
import random
class A(object):
        pass

gc.disable()
a = A()
b = A()
a.x = b
b.x = a
del a
del b
objgraph.show_backrefs(random.choice(objgraph.by_type('A')), max_depth=5, filename='chain.dot')
```

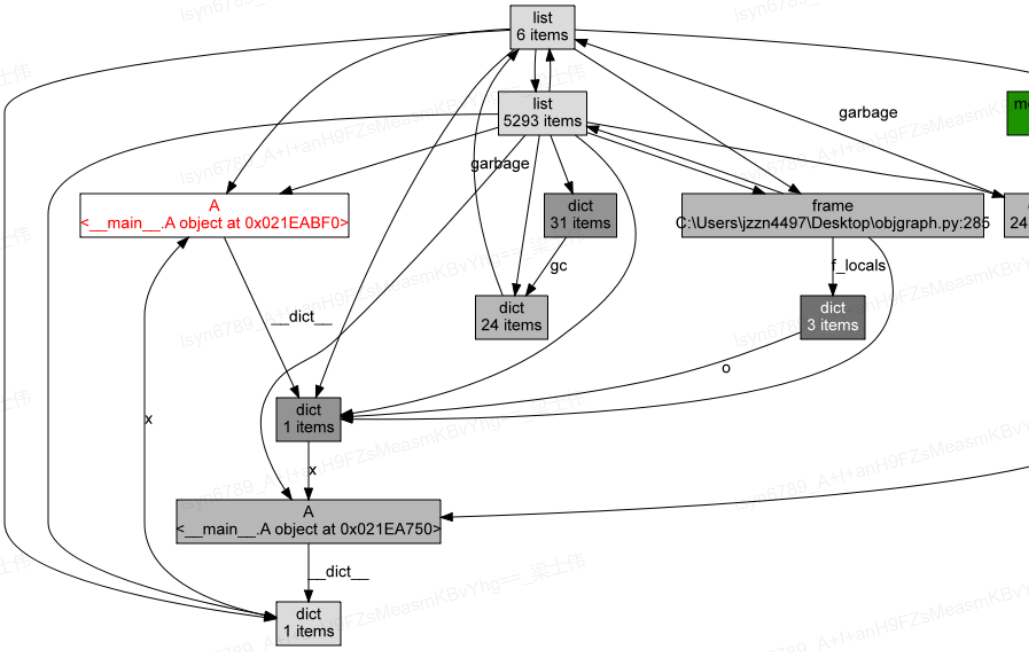

图 4 循环引用示例

示例代码1中展示了将某个循环引用的对象的引用链打印出来，结果如图所示。有时候我们可能不知道哪些对象是发生了循环引用，这里较方便的专门用于寻找循环引用对象的方法，就是利用gc.collect。

示例代码2：

```
import gc
import objgraph
import random
class A(object):
        pass

gc.disable()
a = A()
b = A()
a.x = b
b.x = a
del a
del b

# 此处设置会把被回收的循环引用对象和无法回收的对象都放到gc.garbage中
gc.set_debug(gc.DEBUG_SAVEALL| gc.DEBUG_OBJECTS)
```

6 Tracemalloc

tracemalloc是python 3.4内置的内存分配追踪模块，python 2.X需要打补丁才能使用。与objgraph查找对象分配最多的类型不同，**tracemalloc直接在python内存分配的地方进行hook，以追踪分配内存最多的地方，获取对应的文件、行号和分配调用栈信息。**一般的接口有下面这些：

tracemalloc.start(nframe=1)在python内存分配的地方安装hook，记录下每次内存分配的大小、调用栈等信息。nframe表示记录近的几个栈帧，默认记录最底下的一个栈。

tracemalloc.stop()卸载掉之前安装的hook，清空记录的内存分配信息。

tracemalloc.get_snapshot()获取当前内存分配信息的快照，用于后续对比和分析。

snapshot.statistics(key_type, cumulative=False)获取快照的统计信息，key_type可选参数有'filename'(文件名，以文件为单位的信息)，'lineno'(行号，以文件的行为单位的内存分配信息)，'traceback'(调用栈，获取内存分配信息及对应最近的nframe调用栈)。

snapshot.compare_to(snapshot_instance, key_type)获取和其他快照的对比信息，可以很方便地得知内存的变化情况。

显示分配内存前10的文件行代码示例：

```
# -*- coding: utf-8 -*-
import tracemalloc
tracemalloc.start()
# ... start your application ...
# todo

snapshot = tracemalloc.take_snapshot()
top_stats = snapshot.statistics('lineno')

print("[ Top 10 ]")
for stat in top_stats[:10]:
    print(stat)
```

运行结果：

```
[ Top 10 ]
<frozen importlib._bootstrap>:716: size=4855 KiB, count=39328, average=126 B
<frozen importlib._bootstrap>:284: size=521 KiB, count=3199, average=167 B
/usr/lib/python3.4/collections/__init__.py:368: size=244 KiB, count=2315, average=108 B
/usr/lib/python3.4/unittest/case.py:381: size=185 KiB, count=779, average=243 B
/usr/lib/python3.4/unittest/case.py:402: size=154 KiB, count=378, average=416 B
/usr/lib/python3.4/abc.py:133: size=88.7 KiB, count=347, average=262 B
<frozen importlib._bootstrap>:1446: size=70.4 KiB, count=911, average=79 B
<frozen importlib._bootstrap>:1454: size=52.0 KiB, count=25, average=2131 B
<string>:5: size=49.7 KiB, count=148, average=344 B
/usr/lib/python3.4/sysconfig.py:411: size=48.0 KiB, count=1, average=48.0 KiB
```

内存分配对比示例：

```
# -*- coding: utf-8 -*-
import tracemalloc
tracemalloc.start()
# ... start your application ...
# todo
snapshot1 = tracemalloc.take_snapshot()
# ... call the function leaking memory ...
snapshot2 = tracemalloc.take_snapshot()

top_stats = snapshot2.compare_to(snapshot1, 'lineno')

print("[ Top 10 differences ]")
for stat in top_stats[:10]:
    print(stat)
```

运行结果：

```
[ Top 10 differences ]
<frozen importlib._bootstrap>:716: size=8173 KiB (+4428 KiB), count=71332 (+39369), average=117 B
```

```
/usr/lib/python3.4/unittest/mock.py:491: size=77.7 KiB (+77.7 KiB), count=143 (+143), average=557 B
/usr/lib/python3.4/urllib/parse.py:476: size=71.8 KiB (+71.8 KiB), count=969 (+969), average=76 B
/usr/lib/python3.4/contextlib.py:38: size=67.2 KiB (+67.2 KiB), count=126 (+126), average=546 B
```

7 Memory Profiler

如果只是寻找内存占用的大头和增长迅速的对象类型，前面介绍的objgraph和tracemalloc基本都可以很好地解决了。当你需要聚焦局部化问题时，memory profiler可以让我们了解**函数内部每一行的内存分配**情况。

简单的使用示例：

```
# -*- coding: utf-8 -*-
from memory_profiler import profile

@profile
def my_func():
    a = [1] * (10 ** 6)
    b = [2] * (2 * 10 ** 7)
    del b
    return a

if __name__ == '__main__':
    my_func()
```

运行结果：

```
Line #    Mem usage    Increment   Line Contents
================================================
     4     16.9 MiB     16.9 MiB   @profile
     5                             def my_func():
     6     24.6 MiB      7.6 MiB       a = [1] * (10 ** 6)
     7    177.1 MiB    152.6 MiB       b = [2] * (2 * 10 ** 7)
     8     24.6 MiB      0.0 MiB       del b
     9     24.6 MiB      0.0 MiB       return a
```

如果想了解**随着时间的推移，函数内存分配**情况，可以使用mprof command。将上面的`from memory_profiler import profile`注释
运行以下命令（需要提前pip install matplotlib）：

```
mprof run test.py
mprof plot
```

运行结果：

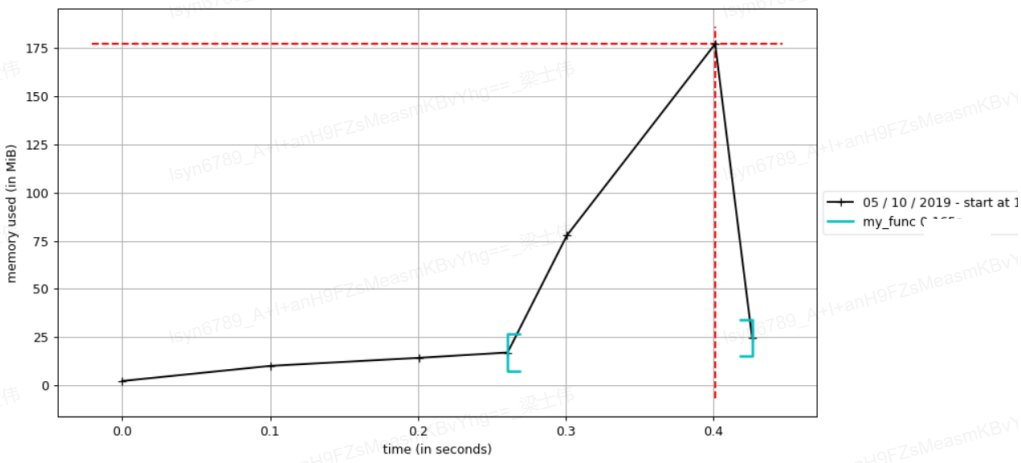

图 5 随时间推移内存分配情况

如果不想用这种命令行的方式获取运行时内存分配情况，可以使用**memory_profiler.memory_usage**，用法如下：

```
    del b
    return a

if __name__ == '__main__':
        # 参数分别为函数对象、位置参数列表、关键字参数，返回函数运行过程中的内存占用列表
    l = memory_usage((my_func, (6,), {'m' : 7}))
    print(l)
```

运行结果：

```
[25.60546875, 25.62109375, 39.59765625, 55.00390625, 69.5390625, 85.06640625, 101.53515625, 116.7578125, 132.076
147.234375, 163.8203125, 178.95703125, 185.84375, 185.84375, 185.84375, 185.84375, 116.42578125, 33.25390625, 25
```

8 Other tools

其他相关的内存分析工具还有：Meliae, Guppy-PE, PySizer, pympler, memprof（基于sys.getsizeof和sys.settrace），Dozer（Cherr
泄漏调试器的WSGI中间件版本）等。

9 Summary

总结一下，在项目中使用python脚本时，关于内存优化这块需要注意：

1）养成良好的习惯，避免一些容易出现内存浪费或内存泄漏的坑。

2）根据项目实际情况，使用前人优化的组件，包括针对数据表优化的各种dict。如有可能，谨慎地在python内部实现上优化一些容器类
分配和使用逻辑。

3）定期使用脚本压测内存的使用情况，及时发现内存占用的异常。当发现有内存问题时，借助对应的工具去追踪、分析内存占用异常或
异常的类型、函数或模块，优先解决对内存影响最大、收益最高的地方。这些内存异常发现和解决的时机当然最好是在开发期，所以及早
测计划、定期执行很重要。当线上出现内存泄漏问题，最好能够在测试机上复现。如果必须要去线上排查，首先要保证不会对在线用户有
（如有需要可以使用开关屏蔽其他用户登录到问题机器或进程上，放出不放进）；其次根据剩余可用内存情况，使用合适的工具去分析，
用那些占用内存大、对性能有影响（容易卡顿的，如guppy）的工具，否则万一把进程搞挂了就直接没有现场了。

全部评论 0

请输入评论内容

(可添加1个视频+5张图片) 匿名

最热 最新

暂无评论

加载完毕,没有更多了